Innovations Syst Softw Eng (2006) 2:65–79
DOI 10.1007/s11334-006-0001-0

ORIGINAL PAPER

Pattern componentization: the factory example

Karine Arnout · Bertrand Meyer

Received: 12 June 2005 / Accepted: 21 September 2005 / Published online: 6 May 2006

Abstract Can Design Patterns be turned into reusable components? To help answer this question, we have performed a systematic study of the standard design patterns. One of the most interesting is Abstract Factory, for which we were indeed able to build a reusable component fulfilling the same needs as the original pattern. This article presents the component's design and its lessons for the general issue of pattern componentization.

1 Patterns and components

In hardly more than a decade, Design Patterns have established themselves as a major resource for effective software design. "*Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to this problem*" [17]. Thanks to the published catalogs of such solutions, starting with [17] itself, software designers can benefit from the wisdom and experience of their predecessors.

K. Arnout
AXA Rosenberg Investment Management LLC,
Barr Rosenberg Research Center, 4 Orinda Way,
Building E, Orinda, CA 94563, USA
e-mail: karnout@axarosenberg.com

B. Meyer (✉)
Computer Science Department, ETH Zurich,
Clausiusstrasse 59, 8092 ETH-Zentrum,
Zurich, Switzerland
e-mail: Bertrand.Meyer@inf.ethz.ch

B. Meyer
Eiffel Software,
356 Storke Road,
Goleta, CA 93117, USA

The idea that we should avoid reinventing the wheel in software construction predates patterns by a long time; *reuse* is a running theme in standard discussions of software engineering. The idea of reuse is to provide software components covering standard needs and make them available through a standard API (Abstract Program Interface[1]) to any program that experiences the corresponding needs.

For all the benefits of design patterns, it is hard to ignore that the idea as usually expressed runs contrary to decades of advances in reuse, which have profoundly improved the state of software development. Pattern advocates themselves often sound skeptical about reuse. The preceding citation was actually truncated: the full sentence states that a pattern describes the core of the solution to the problem "*in such a way that you can use this solution a million times over, without ever doing it the same way twice*". The pattern literature frequently confirms this view that patterns are not reusable components; for example [20]: "*Patterns are not, by definition, fully formalized descriptions. They can't appear as a deliverable*".

Why not? It is hard to accept that patterns, however useful, should force us to step back to pre-reuse times, when sorting an array required finding a solution outline in an algorithms textbook and then adapting it to your needs. Today we use library routines for such tasks. Patterns should provide similarly reusable solution at a higher level of granularity. Inexplicably, the pattern lit-

[1] The original expansion of "API", "Application Program Interface", apparently going back to old IBM software, is no longer meaningful. The acronym itself remains well-understood and relevant. We expand it as "Abstract Program Interface", which captures the idea precisely.

erature rejects the idea, claiming that patterns are somehow of a higher essence than components. We find this view questionable in the absence of concrete evidence and suspect that it may be influenced by the limitations of the programming languages (typically C++ and Java) in which patterns are generally described. Disproving it, at least for some commonly used patterns, would be beneficial, since it is almost always preferable to reuse than to redo: everything else being equal, performance in particular, relying on a reusable component through a well-documented API provides better guarantees of correctness and of general quality than if every developer must code the implementation anew; and it's of course much less effort.

In this view, while patterns as a whole are an admirable advance, a pattern that remains just a pattern is an admission of failure: the failure to abstract the idea to a level where it can be turned into an off-the-shelf solution — rather than studied, understood (or misunderstood), and reimplemented separately by each potential beneficiary. The failure can have various causes:

- Perhaps it is possible to derive a component covering all cases of the pattern, but you did not work hard enough.
- You may be facing a limitation of the programming language you use; another programming language would offer a solution.
- The general assertion (from the pattern literature, as mentioned) that patterns somehow transcend components may hold in the case of a particular pattern.
- For some patterns, you may be able to derive a partial solution in the form of a component that doesn't cover all uses of the pattern, but provides a reusable basis, reducing the amount of specific adaptation work that each user of the pattern must perform.

Patterns have become so important in the practice of software design, and the benefit of reusable solutions over endless individual reimplementation are so compelling, that it is important to examine the principal design patterns in the light of this discussion. We may call the overall goal **componentization**: turning design patterns, whenever possible, into reusable components.

A previous article [28] showed that the "Observer" pattern can be profitably replaced by a simple reusable solution, the *Event Library*, taking advantage of Eiffel mechanisms (genericity, tuples, agents); the benefits including ease of use and greatly improved generality.

Encouraged by that initial success, we set out to perform a systematic analysis [5] of the componentization potential of all the design patterns of [17]. The results include:

- An **analysis** of the challenge of componentization and of the techniques that address it.
- A new **classification** of design patterns in terms of their suitability, or resistance, to componentization.
- The application of componentization techniques to the major design patterns, yielding full or partial componentization in **two thirds of the cases**.
- Concretely, a **Pattern Library** providing reusable implementations of the successfully componentized patterns.
- For the remaining cases, a **Pattern Wizard** facilitating the semi-automatic integration of the patterns into an application.

Both the Pattern Library and the Pattern Wizard are open-source, freely downloadable software available from our download page [16].

We report here on some of the results of this effort, with a special application to one of the best-known pattern: Abstract Factory. (A companion paper [29] details a similar study applied to the Visitor pattern, also resulting in a reusable solution as part of the Pattern Library.) In the rest of this discussion:

- Section 2 presents the componentization effort, describing the criteria we considered to determine whether a pattern is componentizable and the overall results of the study.
- Section 3 presents the intent, advantages and limitations of the Abstract Factory pattern.
- Section 4 describes the design and implementation of our reusable solution: the Factory component of the Pattern Library.
- Section 5 compares the two approaches: use of the Factory Pattern vs. the reusable solution provided by the Factory component of the Pattern Library.
- Section 6 explains the limitations of the componentization approach.
- Section 7 presents some related work about the implementation of design patterns using different programming paradigms and the integration of patterns as programming language features.
- Section 8 draws conclusion about the componentization approach and results.
- Section 9 gives some further research directions.

2 Pattern componentization

Before turning to the specific example of the Abstract Factory pattern, we summarize the componentization study, its assumptions and its results.

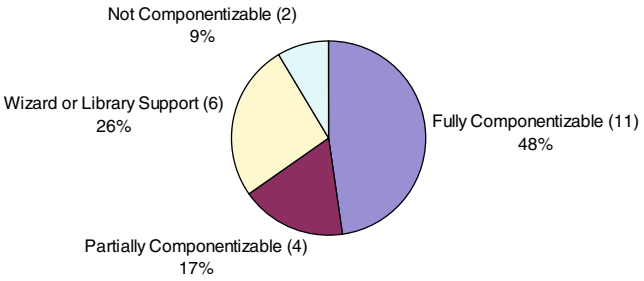

Fig. 1 Componentization results for the patterns in "Design Patterns"

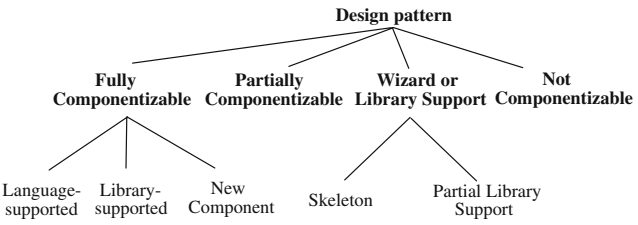

Fig. 2 Pattern componentizability classification

2.1 Overview

Our first pattern analysis, targeting the *Observer* pattern, led to successful componentization through the Event Library [6,28] covering the general idea of publish-subscribe and event-driven development. This provided the basis for the componentization of other patterns including Visitor [29], Composite, Factory as described below, and others all yielding reusable components in the Pattern Library. We confirmed the practical applicability of these components by using them in a number of production applications.

After this first experience we turned to the systematic study of all the 23 patterns in "Design Patterns" [17], the original reference on the topic.

2.2 Overall componentization results

Figure 1 summarizes the results; the precise definition of the categories and the criteria retained are described next.[2]

As the figure indicates:

- For two-thirds (65%) of the original patterns we are able to provide a componentized replacement, enabling application developers to rely on an API from the Pattern Library rather than reimplementing the pattern. More precisely the solution is fully satisfactory in 48% of the cases; in 17% of the cases, it leaves out some cases of the original pattern.
- A quarter of the patterns have "Wizard or library support": we cannot provide a component ready for off-the-shelf use, but we can help through some combination of components addressing part of the problem and the support of the Pattern Wizard to integrate the pattern into an application.
- The remaining 9% are classified as "not componentizable" to reflect that we were not able to make any progress towards a reusable solution. This could of course just reflect our own failure rather than a problem inherent in the patterns themselves.

2.3 Evaluation criteria

In assessing the success of componentization we apply the following criteria:

- *Completeness*: Does the reusable component cover all cases described in the original presentation of the pattern (as given for this study in [17])?
- *Faithfulness*: Does the component provide the same benefits as the original pattern description?
- *Usefulness*: Is it useful to rely on the component rather than merely implementing the pattern?
- *Type-safety*: Is the component type-safe?
- *Performance*: How does the efficiency of using the component compare to a solution that implements the pattern?
- *Extended applicability*: Does the component cover more cases than the pattern?

2.4 Definition of the categories[3]

Figure 2 shows the classification, adapted from the fuller discussion in [5].

The patterns in the first of the top-level categories, **Fully Componentizable** (48% from Fig. 1), are the most interesting result of this study: we can provide an API that fully covers the need for implementing the pattern in an application.[4] There are three subcategories:

- *Language-supported*: no need to do anything at all, the mechanism is already provided by the language. This is the case with the Prototype pattern, for which Eiffel's built-in "clone" facility handles the issue.
- *Library-supported*: in this case the mechanisms of an existing library do the job. This subcategory is

[2] Figure 1 and part of the material in Sect. 2.2 also appear in [29].

[3] Figure 2 and part of the material in Sect. 2.4 also appear in [29].

[4] The inclusion of one pattern, Memento, in the "Fully Componentizable" category is a matter of convention since in that case it is simpler to use the pattern than the component.

Table 1 Eiffel language mechanisms and their use in the Pattern Library

	Prototype	Flyweight	Observer	Mediator	Abstract Factory	Factory Method	Visitor	History-executable Comand	Auto-executable Command	Transparent Composite	Safe Composite	Chain of Responsibility	Two-part Builder	Three-part Builder	Proxy	State	Original strategy	Strategy with agents	Memento
Client/supplier mechanism	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Simple inheritance		X	X				X	X		X	X	X			X	X			
Multiple inheritance		X							X										
Unconstrained genericity		X	X	X	X	X	X	X	X	X	X	X	X	X					
Constrained genericity		X	X	X									X	X	X		X		
Design by contract	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Type conversion																			
Agents		X	X	X	X	X	X	X	X				X	X				X	
Cloning facilities	X																		

included for completeness since we have not so far uncovered any example.
- *New Component*: this is the most original case, indicating that we were able to develop a component that removes the need for the pattern. The Factory family of patterns, discussed below, is an example.

The second category, **Partially Componentizable**, covers patterns for which we were also able to develop a component for the Pattern Library, but the resulting solution is not quite *complete* or *faithful* as defined above.

In the **Wizard or Library Support** category, there is no reusable component in the Pattern Library, but we are able to help developers integrate the pattern into their application through either or both of two techniques:

- They can use the Pattern Wizard to produce a skeleton describing the architecture of the pattern, then fill in the specific elements.
- They can rely on some partial component support from existing libraries.

Patterns of the last category, **Not Componentizable**, have resisted all our efforts: we can neither provide a component, even partial, nor generate a skeleton. These are the patterns truly justifying the reuse-skeptic attitude of the pattern literature highlighted in the citations at the beginning of this article. Fortunately for our effort, only two of the standard design patterns belong to that category: Façade and Interpreter.

2.5 Role of specific language and library mechanisms

The results of componentization are clearly dependent on the target programming language. The effort reported here benefits from the mechanisms of Eiffel [26, 30], as described in the corresponding ECMA standard [14]. The following language capabilities play a particularly important role; Table 1 shows their actual use in the Pattern Library.

- **Genericity**: a basic Eiffel facility for defining classes parameterized by types, and already used in all Eiffel data structure libraries. Genericity is *constrained* if the actual generic parameter must be a descendant of a specific type, *unconstrained* otherwise. All the Pattern Library classes (for the fully componentized patterns) rely on unconstrained genericity;

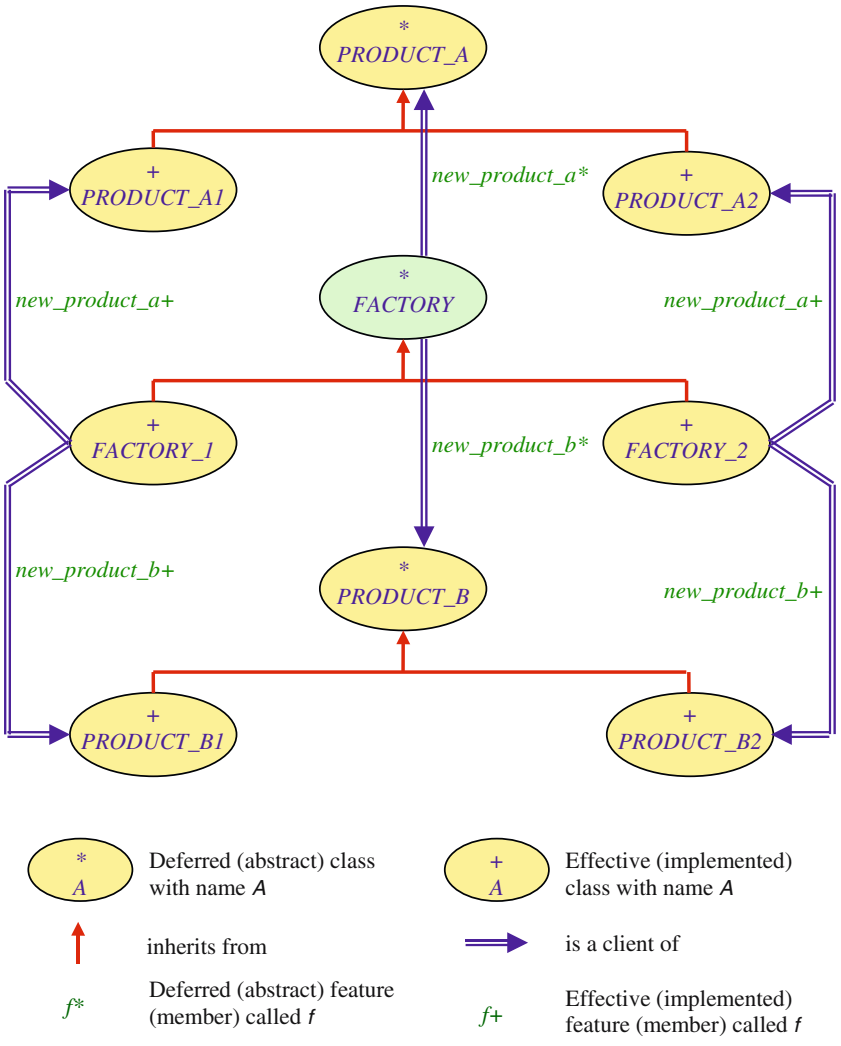

Fig. 3 Class structure for the Abstract Factory pattern

three (Observer, Mediator and Flyweight) use the constrained form.

- **Agents** [12]: objects encapsulating routines ready to be called. 73% (8 out of 11) of the fully componentized patterns use agents.
- **Tuples**: sequences of values of set types, similar to classes but anonymous.
- Support for **Design by Contract**™ [23–25] to equip the componentized patterns with precise semantic properties. All Pattern Library classes take advantage of contracts.
- **Client-supplier** relationships.
- Single and multiple **inheritance**.
- Automatic **type conversion**, which exists in all languages for basic types (as in converting from integers to reals) but benefit in Eiffel from a full-fledged mechanism applicable to any user-defined type and carefully combined with inheritance [28]. This facility is mentioned here even though it is not used in any of the currently componentized patterns; informal investigations of other patterns show that it can play a useful role.
- **Cloning**: built-in facilities for duplicating objects.

3 "Abstract factory" as a pattern

The Abstract Factory design pattern is a widely used solution to create object families without specifying the concrete type of each object. In this section we describe the pattern, its benefits and limitations.

3.1 Pattern description

The Abstract Factory pattern is intended to "*provide an interface for creating families of related or dependent objects without specifying their concrete classes*" [17]. Figure 3 (using, as other class diagrams in this article, the conventions of the BON method [32], explained by the legend below) shows the classes involved and their relationships.

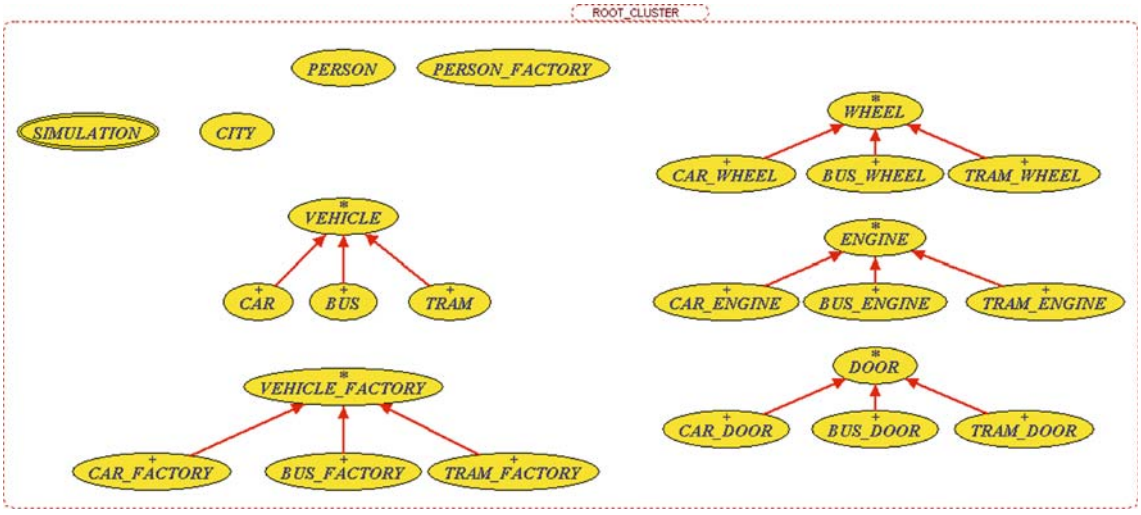

Fig. 4 Traffic simulation with the Abstract Factory pattern

In this example and others, some of the names and conventions have been changed from [17] for consistency with the rest of the discussion, but this does not affect the substance of the patterns. A feature is "effective" if it is implemented, "deferred" if it is only specified (with a contract if applicable). A class is effective if all its features are effective, deferred otherwise. A deferred class does not have to be fully abstract: it may contain a mix of deferred and effective features.

The deferred class *FACTORY* declares the deferred factory functions, *new_ product_a* and *new_ product_b*, which create and return anew instance of *PRODUCT_A* and *PRODUCT_B*. These functions are effected (made effective) in *FACTORY_1* and *FACTORY_2*, in a covariant way: in *FACTORY_1*, *new_ product_a* returns an instance of *PRODUCT_A1* and *new_ product_b* of *PRODUCT_B1*; in *FACTORY_2*, they produce a *PRODUCT_A2* or a *PRODUCT_B2*.

3.2 Using the pattern

Because the factory pattern is just a pattern, the above software structure must be instantiated anew for each application. Here is such an example, covering a traffic simulation system. Cities have people and vehicles; we will use factories for both of these kinds of objects. A vehicle – car, bus, tram – has an engine, wheels, and doors, which vary with the vehicle type.

Figure 4 shows the structure. The *_FACTORY* classes implement the pattern; the others describe the application domain. To create a new person, clients will use the factory class *PERSON_FACTORY*; to create a vehicle, they use *VEHICLE_FACTORY*, deferred but with three effective descendants *CAR_FACTORY*, *BUS_FACTORY* and *TRAM_FACTORY*. Note how these mirror the *VEHICLE* hierarchy.

The example has been spelled out, including classes such as *VEHICLE*, *CAR*, *WHEEL* and *CAR_WHEEL* which have no direct relation to the pattern but help make this application of the pattern complete and realistic. Here are possible versions of these classes, starting with *VEHICLE*[5]:

```
note
        description: "General notion of vehicle for traffic simulation"
deferred class
        VEHICLE
feature {NONE} -- Initialization
        make (e: like engine; w: like wheels; d:like doors)
                                -- Set engine to e, wheels to w, doors to d.
                require
                                engine_exists: e /= Void
                                wheels_exists: w /= Void
                                wheels_does_not_contain_void: not w.has (Void)
                                wheel_count_positive: w.count > 0
                                wheels_valid: w.count = wheel_count
                                                and w.count = w.capacity
                                doors_exists: d /= Void
                                doors_does_not_contain_void: not d.has (Void)
                                doors_valid: d.count = door_count and
                                                d.count = d.capacity
                do
                                engine := e ; wheels := w ; doors := d
                ensure
                                engine_set: engine = e
                                wheels_set: wheels = w
                                doors_set: doors = d
                end
```

[5] All the program examples follow the Eiffel standard [14].

```
feature -- Access
        engine: ENGINE
                        -- Engine

        wheels: ARRAYED_LIST [WHEEL]
                        -- Wheels

        doors: ARRAYED_LIST [DOOR]
                        -- Doors

        wheel_count: INTEGER
                        -- Number of wheels
                deferred
                end
        door_count: INTEGER
                        -- Number of doors
                deferred
                end
invariant
        engine_exists: engine /= Void
        doors_exists: doors /= Void
        wheels_exists: wheels /= Void
        doors_does_not_contain_void: not doors.has (Void)
        wheels_does_not_contain_void: not wheels.has (Void)
        wheel_count_positive: wheel_count > 0
        door_count_valid: door_count = doors.capacity and
              doors.count = door_count
        wheel_count_valid: wheel_count = wheels.capacity and
              wheels.count = wheel_count
end
```

A typical descendant of *VEHICLE*:

```
note
        description: "General notion of car for traffic simulation"
class
        CAR
inherit
        VEHICLE
                redefine engine, wheels, doors end
create
        make

feature -- Access
        engine: CAR_ENGINE
                        -- Engine

        wheels: ARRAYED_LIST [CAR_WHEEL]
                        -- Wheels

        doors: ARRAYED_LIST [CAR_DOOR]
                        -- Doors

        Wheel_count: INTEGER = 4 ; Door_count: INTEGER = 4
                        -- Number of wheels and doors

invariant
        four_wheels: wheel_count = 4
        four_doors: door_count = 4
end
```

Other descendants include *BUS*, *TRAIN* etc. We also need wheels:

```
note
        description: "General notion of wheel for traffic simulation"
deferred class
        WHEEL
feature -- Initialization
        make (d: like diameter)
                        -- Set diameter to d.
                require
                        diameter_valid: d >= minimum_diameter
                                and d <= maximum_diameter
                do
                        diameter := d
                ensure
                        diameter_set: diameter = d
                end
feature -- Access
        diameter: INTEGER
                        -- Diameter (in millimeters)

        minimum_diameter: INTEGER
                        -- Minimum diameter (in millimeters)
                deferred
                end

        maximum_diameter: INTEGER
                        -- Maximum diameter (in millimeters)
                deferred
                end
feature -- Status report
        is_valid: BOOLEAN
                        -- Is wheel meaningful?
                do
                        Result := (diameter >= minimum_diameter and
                                diameter <= maximum_diameter)
                ensure
                        definition: Result = (diameter >=
                              minimum_diameter and diameter <= maximum_diameter)
                end
invariant
        minimum_diameter_positive: minimum_diameter > 0
        min_and_max_valid: minimum_diameter <=  maximum_diameter
        ivalid: is_valid
end
```

Descendants of *WHEEL* may include *CAR_WHEEL* etc.

Now the factory classes. *VEHICLE_FACTORY*, deferred, contains the factory operations such as

```
new_vehicle (p,d,w,h: INTEGER): VEHICLE
                -- New vehicle with engine power p,  wheel diameter d,
                -- door width w and door height h
        require
                power_valid: p >= minimum_power and p <= maximum_power
                diameter_valid: d >= minimum_diameter and
                        d <= maximum_diameter
                width_valid: w >= minimum_width and w <= maximum_width
                height_valid: h >= minimum_height and
                        h <= maximum_height
        deferred
        ensure
                vehicle_exists: Result /= Void
        end
```

The contracts use features such as *minimum_ power*, declared in class *VEHICLE_FACTORY* as deferred and made effective in the descendants; these features will in practice have to duplicate code that exists in classes *ENGINE*, *WHEEL*, and *DOOR*.

Descendants of *VEHICLE_FACTORY*, such as *CAR_FACTORY*, define their own factory features, this time effective, for example:

```
new_car (p, d, w, h: INTEGER): CAR
                -- New car with engine power p, wheel diameter d,
                -- door width w and door height h
        do
                create Result.make (
                    new_engine (p), new_wheels (d), new_doors (w, h))
        end
```

new_engine is another factory feature declared in *VEHICLE_FACTORY*. *new_wheels* and *new_doors* are not part of the class interface, but deal with the implementation; they use the factory features *new_wheel* and *new_door*.

Using the *Abstract Factory* pattern, the root creation procedure (feature *make* of class *SIMULATION*) has such calls as:

```
zurich_city.vehicles.extend (car_factory.new_car (
            power, wheel_diameter, door_width, door_height))
```

We can define *car_ factory* as a once function (creating an object on first call, then returning on every subsequent call a reference to that object):

```
car_factory: CAR_FACTORY
                -- Car factory object, shared
        once
                create Result
        ensure
                car_factory_exists: Result /= Void
        end
```

This technique ensures that we have exactly one "car factory" object in the system, and is applied to other factories as well. The rest of the factory-based part of the traffic simulation system follows directly from the above models.

3.3 Abstract Factory as a pattern: an analysis

From a software engineering perspective, the Abstract Factory pattern lends itself to criticism on two grounds:

- It causes considerable code redundancy; remember in particular that (in the general model of Sect. 3.1) classes *FACTORY_1*, *PRODUCT_A* and *PRODUCT_A1* are templates for m, n and $m * n$ classes respectively, where m is the number of factories (two in this example, 1 and 2) and n the number of products (also two, A and B). [17] acknowledges this problem by noting that the pattern "*requires a new concrete factory subclass for each product family, even if the product family differs only slightly*".
  It is well known that code duplication is the source of considerable trouble in software construction and maintenance. Future changes made to one variant must be carried over to the others; ditto for bug corrections. The software becomes uselessly complex, raising new challenges for project and configuration management.
- Another problem is the solution's lack of flexibility. The deferred class *FACTORY* must specify a fixed number of factory functions and their signatures. As a consequence, "*supporting new kinds of products is difficult*": introducing a new family of products requires changing class *FACTORY* and all its descendants. No wonder this leads to conclusions that patterns are inherently non-componentizable.
  To address these issues, [17] suggests combining Abstract Factory with the *Prototype* pattern. In Eiffel there is no need for this pattern, as its purpose, producing *clones* of objects, is directly addressed by a built-in language and library mechanism.

Combining cloning with **genericity** leads to the basic idea behind the reusable solution – the Factory components of the Pattern Library.

4 "Abstract Factory" as a component

We now examine the reusable solution devised for the Abstract Factory pattern.

A reusable component lives or dies by its API – by how easy it is for client programmers to take advantage of the component through the purely abstract description of its interface. For this reason we first illustrate the Abstract Factory component through a typical example of its *use*; next we study how this solution compares, for the application developer, with implementing the pattern directly; then we look at the component's internal *design and implementation*.

4.1 Using the Abstract Factory component

Instead of having to write one factory class per vehicle type, users of the Pattern Library's Abstract Factory component rely on a single generic class *FACTORY*[*G*]. This is the only class we need to examine.

The root creation procedure *make* is similar to its original version: the difference is that instead of calls such as

```
zurich_city.vehicles.extend (car_factory.new_car
                        (power, wheel_diameter,  door_width, door_height))
```

it suffices to call the factory function *new_with_args* from class *FACTORY*[*G*]:

```
zurich_city.vehicles.extend (car_factory.new_with_args (
            [car_power, car_wheel_diameter, car_door_width, car_door_height]))
```

Function *new_with_args* returning a new instance of *G* – the generic parameter type of *FACTORY* – created by calling the creation procedures of *G* with the argument given. The function indeed takes a single argument, an Eiffel **tuple**, given by a list of values in square brackets. The function *car_ factory* is defined simply as

```
car_factory: FACTORY [CAR]
                                    -- Car factory
        once
                create Result.make (agent new_car)
        ensure
                car_factory_created: Result /= Void
        end
```

and *new_car* as

```
new_car (p, d, w, h: INTEGER): CAR
                -- New car with power engine p, wheel diameter d,
                -- door width w and door height h
        require
                power_valid: p >= {CAR_ENGINE}.minimum_power and
                        p <= {CAR_ENGINE}.maximum_power
                diameter_valid: d >= {CAR_WHEEL}.minimum_diameter and
                        d <= {CAR_WHEEL}.maximum_diameter
                width_valid: w >= {CAR_DOOR}.minimum_width and
                        w <= {CAR_DOOR}.maximum_width
                height_valid: h >= {CAR_DOOR}.minimum_height and
                        h <= {CAR_DOOR}.maximum_height
        do
                create Result.make
                        (car_engine_factory.new_with_args ([p]),
                        new_car_wheels (d), new_car_doors (w, h))
        ensure
                car_exists: Result /= Void
        end
```

The notation {*C*}.*m* yields the value of a constant *m* declared in a class *C*.

4.2 Discussion: pattern versus reusable component

On the basis of the API (even though we have not seen the implementation yet) we can now compare, from a client's perspective, the component-based solution against the original direct implementation of the pattern. The traffic simulation program provides an appropriate example.

On the negative side, the new solution loses some flexibility: it no longer uses specific factory classes such as *CAR_FACTORY* and *BUS_FACTORY* inheriting from a common ancestor *VEHICLE_FACTORY*; the equivalent code is in a single *SIMULATION* class. This may cause some code redundancy, for example between features *new_car*, *new_bus*, and *new_tram*; in addition class *SIMULATION* can be bulky.

On the positive side we note:

- Reusability: the single remaining factory class, *FACTORY* [*G*] is a library class that can be reused in many applications whereas classes such as *CAR_ FACTORY* were specific to one application and could not be reused without considerable changes.
- Ease of use: although simplicity of an API is partly a matter of opinion, we think the traffic simulation example demonstrates that the Pattern Library makes it particularly easy to equip any application with abstract factories.
- Fewer classes: there is now just one factory class, the general-purpose *FACTORY* [*G*], instead of five (*VEHICLE_FACTORY* and one for each other type of vehicle).
- No code duplication for contracts: there is no more need to duplicate in the root class *SIMULATION* the constant features *minimum_ power*, *maximum_ power* etc. from *ENGINE*, *WHEEL*, and *DOOR*, since we can now just use {*CAR_ENGINE*}. *minimum_ power* etc. This was not possible in the original version since {**like** *new_engine*}. *minimum_ power* is not a valid notation (a language limitation which conceptually seems impossible to remove).

4.3 Towards a reusable component

We now examine the internal design of the Abstract Factory component of the Pattern Library. Before presenting the final version, which involves several advanced language mechanisms such as constrained genericity and agents, we briefly present a few intermediate attempts and show why they were not completely satisfactory. The reader who is only interested in the final version can skip to Sect. 4.4.

In a first approach, the factory function *new* returns a new instance of *G* by cloning a **prototype**, through the built-in function *cloned* coming in Eiffel from the universal ancestor class *ANY* and hence available to all classes. (In earlier versions it was known as *clone* or *twin*.)

```
note
        description: "[
                Mechanisms for creating objects of type 'G' by shallow cloning of a prototype.
                ]"
        version: "Version 1, not final"
class
        FACTORY [G]
create
        make
feature -- Initialization
        make (p: like prototype)
                                    -- Set prototype to p.
                require
                        prototype_exists: p /= Void
                do
                        prototype := p
                ensure
                        prototype_set: prototype = p
                end
```

```
feature -- Factory function
        new: G
                        -- New instance of type G
                do
                        Result := prototype.cloned
                ensure
                        Result_exists: Result /= Void
                end
feature {NONE} -- Implementation
        prototype: G
                        -- Prototype from which new objects are created
invariant
        prototype_exists: prototype /= Void
end
```

Function *new* uses shallow cloning, as provided by the library feature *cloned*. It is possible to use deep cloning instead (see [5] for more details on this and other variants). To define actual factory classes we provide actual generic parameters, as in *FACTORY*[*VEHICLE*].

This solution, however, does not provide a direct way to initialize newly created objects. For this we should rely on Eiffel's **constrained genericity**: we force the generic parameter of *FACTORY*[*G*] to provide *default_create* as creation procedure; Eiffel rules imply that in this case the creation instruction **create** *x* (without an explicit creation procedure) is valid for *x* of type *G*; it will call as creation procedure the version of *default_create* corresponding to the actual generic parameter. The class becomes just:

```
note
        description: "Object factory"
        version: "Version 2, not final"
class
        FACTORY [G -> ANY create default_create end]
feature -- Factory
        new: G
                        -- Instantiate a new object of type G.
                do
                        create Result
                ensure
                        new_instance_exists: Result /= Void
                end
end
```

In this version *FACTORY*[*PRODUCT*] is only valid if class *PRODUCT* lists feature *default_create* as one of its creation procedures. As a consequence, class *PRODUCT* has to be an effective (non-deferred) class.

Whenever we need a product, we call the feature *new* on the appropriate factory instead of creating the object directly. As in the original pattern, this design helps separate the object creations from the application logic. There is no more need for defining a prototype to be cloned.

A number of drawbacks remain. In our example we need factories of types *FACTORY* [*CAR*], *FACTORY* [*BUS*] etc. A typical one reads:

```
car_factory: FACTORY [CAR]
                        -- Car factory
        once
                        create Result
        ensure
                        factory_exists: Result /= Void
        end
```

(using a once function to guarantee sharing). *CAR* must now provide *default_create* as a creation procedure. There is no reason it did before since this is just an ordinary application class. Even if we accept the prospect of modifying the class text, this could break a class invariant clause such as

```
engine_exists: engine /= Void
```

which any creation procedure must ensure. This requires redefining *default_create* in *CAR* :

```
default_create
                        -- Set up maze.
        do
                        create engine.make (...)
        end
```

This scheme may be made to work in this particular case, but it does not generalize to classes with more sophisticated invariants and creation procedures that (correspondingly) require arguments. Updating such classes to make them usable as actual generic parameters for *FACTORY* would break their existing clients. In any case the prospect of modifying existing classes from the application domain just to enable them to participate in factories is not practical.

All this indicates that the reusable solution as obtained so far does not scale up.

The root of the remaining problem is that the solution as obtained so far requires creation to be specified statically. We can make the solution more dynamic thanks to agents.

4.4 The factory component: using agents

An agent in Eiffel is an object that wraps a routine; for a known routine *r*, **agent** *r* defines the associated agent; another routine to which this agent has been passed as the argument *a* does not need to know what the original *r* was, but will be able to call it anyway through the *call* feature available on all agents, in the form *a.call* ([...]), where the argument is a tuple.

By passing a creation procedure to the factory as an agent, we can wait until run time to provide every factory object with its tailor-made initialization mechanism.

This observation gives us the final version of the class *FACTORY* as used in the Pattern Library.

```
note
        description: "Object factory"
        version: "Final version from the Pattern Library"
class FACTORY [G] create
        make
feature -- Initialization
        make (func: like factory_function)
                                -- Set factory_function to func.
                require
                                func_exists: func /= Void
                do
                                set_factory_function (func)
                ensure
                                function_set: factory_function = func
                end
feature -- Access

        factory_function: FUNCTION [ANY, TUPLE, G]
                                -- Factory function creating instances of G

feature -- Factory functions
        new: G
                                -- New instance of G
                do
                                factory_function.call ([])
                                Result := factory_function.last_result
                ensure
                                new_exists: Result /= Void
                end

        new_with_args (args: TUPLE): G
                                -- New instance of type G initialized with args
                require
                                valid: factory_function.valid_operands (args)
                do
                                factory_function.call (args)
                                Result := factory_function.last_result
                ensure
                                new_exists: Result /= Void
                end

feature -- Element change
        set_factory_function (func: like factory_function)
                                -- Set factory_function to func.
                require
                                func_exists: func /= Void
                do
                                factory_function := func
                ensure
                                function_set: factory_function = func
                end
invariant
        factory_function_exists: factory_function /= Void
end
```

4.5 Component properties

We note the following properties of the componentization of the Abstract Factory pattern using the criteria of Sect. 2.5:

- *Completeness*: The Factory component covers all cases described in the original Abstract Factory pattern. An apparent limitation is that it is possible to create only one kind of product; but this is simply a matter of convention: if you need to create two kinds of product, you'll just use two factories.
- *Usefulness*: The Factory component can indeed be used in practice as an effective replacement for the pattern.
- *Faithfulness*: While the architecture is different, the Factory component retains the intent and spirit of the original Abstract Factory pattern.
- *Type-safety*: The Factory component mainly relies on type-safe mechanisms of constrained genericity and agents.
- *Performance*: The main difference between the internal implementation of the Factory component and the Abstract Factory design pattern is the use of agent calls instead of direct calls to factory functions. Agents carry a performance overhead, but that overhead is very small on the overall application. Our benchmarks on a typical application show a degradation of only 7%.
- *Extended applicability*: The Factory component does not cover more cases than the original Abstract Factory pattern.

5 Limitations of the approach

We have shown on the example of Abstract Factory (confirmed by many others in our study) that it is possible to turn that design patterns into reusable components. There are, however, some limitations to this approach.

5.1 One pattern, several implementations

The first limitation does not apply to the example of this article but to the approach as a whole: not all patterns appear fully componentizable. Counter-examples include "State", "Builder" and "Proxy". So we will in the short term continue to need some patterns that are only patterns, not components.

5.2 Language dependency

We heavily rely on Eiffel mechanisms. We have not tried to transpose the approach to other languages. As an initial assessment for two recent languages:

- Genericity plays an important role and until recently was specific to Eiffel (and C++ with its macro-like "template" mechanism). Both Java and C# [21] are in the process of adding a generic mechanism, which will help.
- Neither Java nor C# support multiple inheritance except from interfaces; this precludes the direct

imitation of the Eiffel solutions using multiple inheritance.
- Neither Java nor C# support contracts; this affects the clarity of reusable solutions and the ease of making arguments supporting their correctness, but not the architecture per se. We may point here to our earlier work [2, 3] on automatically extracting contracts from non-contracted classes, for example on .NET, which may prove useful here.
- C#'s "delegates" are a more limited form of agents, which may be applicable to the many Pattern Library solutions relying on agents.
- Java has explicitly rejected any form of agents or delegates. This puts into question the applicability to Java of most of the Pattern Library ideas (although reflection might provide some solutions).

5.3 Usefulness

Some programmers may prefer to write their own customized design pattern implementation for any of the following reasons:

- *Usage complexity*: In some cases, using the reusable component may be less user-friendly than a customized pattern implementation. It may also be somewhat overkill when the pattern implementation is very simple. The Memento pattern is an example of this case.
- *Performance*: Some componentized versions of design patterns imply a performance overhead compared to a "traditional" pattern implementation. This is not the case with Abstract Factory, but for example the componentized Visitor may imply an overhead of 30–50% over direct use of the pattern [29]. Although the pattern typically accounts for only part of the execution time of an entire application, this overhead may be intolerable in some performance-critical cases.

6 Related work

6.1 C++ implementation of design patterns

Alexandrescu explains [1] how to implement some design patterns in C++. Although related, his work only addresses a few design patterns, and its focus is different: providing different implementations of the patterns, many of them relying on C++ templates, rather than reusable (componentized) solutions.

6.2 Aspect implementation of design patterns

Hannemann and Kiczales [19, henceforth "H & K"] explored how to take advantage of aspect-oriented programming (AOP) [22] to implement aspects; they implemented the same 23 patterns as our study, in both Java and AspectJ [13], an aspect-oriented extension for Java. They evaluated the resulting code according to four properties:

- *Locality*: The pattern code is confined in aspects; it does not extend to existing classes participating in the pattern.
- *Reusability*: The abstract aspect can be reused (programmers still need to write concrete aspects).
- *Composition transparency*: Some classes can be involved in many patterns transparently (because the pattern code is located in an aspect and does not touch the participant classes).
- *(Un)pluggability*: Adding or removing a pattern is easy because participant classes do not know about their involvement in the pattern implementation.

Using AspectJ sometimes came down to an implementation change and sometimes resulted in a completely new design structure.

The reusability classification of the aspect implementations is the most closely related to this work. While H & K's definition of reusability differs from the one presented in this article (abstract aspects vs. concrete classes), it is interesting to see the similarities.

H & K note that Observer code usually spreads across several classes, making maintenance harder. For example, concrete subjects are likely to have many similar features which call a procedure *update_observers*. Using aspects solves the problem through the notion of pointcut: one can define a set of points in the program execution where the feature *update_observers* needs to be called – no need to pollute the code of all concrete subjects anymore. As a result H & K categorize Observer as reusable with AspectJ. They found eleven other patterns for which "*a core part of the implementation can be abstracted into reusable code*". Comparing these results with our componentizability classification:

- Our classification agrees on ten of their twelve reusable patterns. The Singleton and Iterator patterns resisted componentization work; note however that Iterator is already supported to some extent by existing Eiffel libraries and that the notion of "frozen class" now introduced in standard Eiffel makes it possible to generate skeleton classes for Singleton.

- For Proxy, Builder and State we achieve partial componentization. The difference simply reflects that our classification is more fine-grained; H & K only consider "yes" or "no" answers.
- H & K's results did not succeed in handling Abstract Factory and Factory Method through AspectJ aspects; we were able to componentized them thanks to Eiffel's genericity and agents.
- Both classifications find Adapter, Decorator, Template Method, Bridge, Interpreter and Façade not to be componentizable.

H & K explain their results by the nature of design patterns. They distinguish between patterns with *defining roles* (classes participating in the pattern have no functionality outside the pattern) and those with *superimposing role* (participating classes have outside functionality) and state that most reusability improvements concern patterns of the second category, since superimposed pattern behavior can be moved into an independent reusable module.

In addition to H & K there is considerable activity in the area of applying aspects to patterns; see for example [18]. The expected advantages include a reduction of the number of pattern participants (typically one aspect instead of several classes); better traceability of the code and hence better documentation; more localized pattern code; and more reuse. We may note, however the following limitations:

- Just as our componentization work depends on the programming language used to write the components, AOP approaches depend on the choice of aspect language. For example [18] mentions the difficulty of translating code from AspectJ to HyperJ.
- An aspect implementation typically introduces (as also pointed out by [18]) many small aspects, which are necessary to understand the design. As a result it is not so clear to us what exactly is gained over a standard pattern implementation. With full componentization, the pattern implementation resides entirely in a class or a few classes from a library, understandable through the sole provision of its API.

6.3 Language support of design patterns

Chambers et al. write [11] that design patterns "*have proved so useful that some have called for their promotion to programming language features*". As an example, Bosch describes [8] a new language called LayOM with original support for design patterns through such constructs as "layer" and "state", which permit to represent patterns.

Clearly, the inability to componentize patterns fully is an interesting source of language design ideas. An example is the introduction of "frozen" classes into standard Eiffel, which was motivated in part by the need to support better componentization of Singleton [4]. But to avoid what Chambers et al. call the "*kitchen sink problem*" one cannot add a language feature for every need. The spirit behind Eiffel's design is that every new functionality should add a significant power of expressiveness to the language at low cost on the overall language complexity, avoiding "featurism" and keeping instead a "*high signal to noise ratio*" [14,26, 30]. A programming language is, in any case, a complex engineering construction; while it is possible and pleasurable to play with tentative language constructs in an academic environment with the hope of influencing future industrial languages, the industrial languages themselves tend to evolve slowly, and the addition of any new concept is a major endeavor that must be reconciled with many engineering criteria: backward compatibility, consistency with other language constructs, implementability at reasonable cost, possible performance hits (compile-time and run-time), teachability, insertion into release schedules etc. So we cannot envision turning every great language design idea, however attractive on paper, into a realistic language feature.

Componentization, whenever applicable, seems the more desirable approach. Only when it fails for reasons that appear fundamental (rather than lack of insight on the part of those attempting it) should one turn to the investigation of possible language extensions.

6.4 Automatic code generation from patterns

Budinsky et al. [9] describe a tool (dating back to as early as 1996) for generating code from design patterns; this is a precursor to our Pattern Wizard. There are, however, important differences. Some are of implementation (HTML browser and Perl scripts instead of an object-oriented design, generation of C++ rather than Eiffel). More fundamentally: our Pattern Wizard is simpler to use and meant solely to complement the use of the Pattern Library.

The "Presenter" part of the tool by Budinsky et al., on the other hand, provides more options and in general more flexibility, from which the Pattern Wizard could benefit.

7 Conclusion

The goal of this work was to explore a conjecture from [27]: "*A successful pattern cannot just be a book*

description: it must be a software component, or a set of components"; see also Pinto et al. [31]: "*The Design Patterns fail providing a solution because it is necessary to apply and implement the same design pattern over and over, for each component*". The results obtained so far show that, for a large part, patterns can indeed be replaced by components. The success ratio cited in 2.2 (48% full componentization, 17% partial) are encouraging. The Abstract Factory example shows the process at work.

The results of componentization – the classes in the Pattern Library – are of good quality: type-safe and contract-equipped. Significant practical usage, including in industrial applications, has demonstrated their practicality.

Another directly usable result is the componentizability classification, which gives programmers a reference to know where to look for help: in the best case, just go to the applicable API and do not bother any further; otherwise, use the Pattern Wizard if applicable; in the couple of remaining cases, you know you have no one to turn to but yourself.

The progress of software engineering suggests that it is usually better to reuse than to redo; the componentization results shows that, for design patterns, it is often possible to use the better alternative.

8 Future work

The following directions appear interesting for continuation of this work.

- *Componentizing more design patterns*: there are plenty of patterns beyond those we studied so far; see e.g. [10].
- *Testing componentized patterns*: testing the reusable components resulting from pattern componentization is essential because reuse increases both good and bad aspects of the software. Robert Binder explains that "*components offered for reuse should be highly reliable; extensive testing is warranted when reuse is intended*" [7]. The AutoTest environment [15] supports the automatic testing of contracted components and could be fruitfully applied to the results of pattern componentization.

Acknowledgements We are grateful to Éric Bezault for comments on this article and the componentization work in general.

References

1. Alexandrescu A (2001) Modern C++ design, generic programming and design patterns applied. Addison-Wesley, Reading
2. Arnout K, Meyer B (2003) Finding Implicit Contracts in .NET Components. In: de Boer F, Bonsangue M, Graf S, de Roever WP (eds) Proceedings of FMCO 2002 (Ist international symposium on formal methods for components and objects, Leiden, The Netherlands, November 2002), LNCS vol 2852. Springer, Berlin Heidelberg New York, pp 285–318,
3. Arnout K, Meyer B (2003) Uncoverring hidden contracts: the .NET example. IEEE Comput 36(11):48–55
4. Arnout K, Bezault E (2004) How to get a Singleton in Eiffel? In: Proceedings of TOOLS USA 2003 (44th international conference on the technology of object-oriented languages and systems), Santa Barbara, CA, September 2003, in J Object Technol 3(4), 75–95. Available (March 2006) at http://www.jot.fm/issues/issue_2004_04/article5
5. Arnout K (2004) From patterns to components, PhD thesis, ETH Zurich. Available (2006) at http://se.ethz.ch/people/arnout/patterns/
6. Arslan V, Nienaltowski P, Arnout K (2003) An object-oriented library for event-driven design. In: Böszörmenyi L, Schojer P (eds) Proceedings of JMLC (joint modular languages conference), Klagenfurt, Austria, August 2003. LNCS 2789. Springer, Berlin Heidelberg New York, pp 174–183
7. Binder RV (1999) Testing object-oriented systems, models, patterns and tools. Addison-Wesley, Reading
8. Bosch J (1998) Design patterns as language constructs. J Object Oriented Programm 11(2): 18–32
9. Budinski F, Finnie M, Yu P, Vlissides J (1996) Automatic code generation from design patterns. IBM Syst J, 35(2):151–171. Available (March 2006) at http://www.research.ibm.com/designpatterns/pubs/codegen.pdf
10. Bushmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented software architecture: A system of patterns, Vol 1. Wiley, New York
11. Chambers C, Harrison B, Vlissides J (2000) A debate on language and tool support for design patterns. In Proceedings of POPL 2000 (27th ACM SIGPLAN-SIGACT symposium on principles of programming languages), Boston, ACM Press, pp 277–289
12. Dubois P, Howard M, Meyer B, Schweitzer M, Stapf E (1999) From calls to agents. JOOP (J Object Oriented Programm), 12(6): 1999. Available (March 2006) at http://www.inf.ethz.ch/~meyer/publications/joop/agent.pdf
13. Eclipse (2006) Aspectj project. Available (March 2006) at http://www.eclipse.org/aspectj
14. ECMA International (2005) ECMA Standard 367: Eiffel: analysis, design and programming language, approved 21 June 2005. (Expected to become ISO standard in 2006)
15. ETH Zurich (2006) Chair of Software Engineering: AutoTest project pages and articles, available (March 2006) from http://se.ethz.ch/people/leitner/auto_test/
16. ETH Zurich (2006) Chair of software engineering: pattern library and pattern wizard; available (March 2006) from the download page. http://se.ethz.ch/download
17. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns. Addison-Wesley, Reading
18. Hachani O, Bardou D (2003) On Aspect-oriented technology and object-oriented design patterns. In: Proceedings of the workshop on Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003 (17th European conference for object-oriented programming), Darmstadt, Germany. Available (March 2006) at www.comp.lancs.ac.uk/computing/users/chitchya/AAOS2003/Assets/hachani_bardou.pdf
19. Hannemann J, Kiczales G (2002) Design pattern implementation in Java and AspectJ. In: OOPSLA 2002 (17th ACM conference on object-oriented programming, systems, languages and applications, Seattle 4–8 November 2002, ACM Press, pp 161–173

20. Jézéquel J-M, Train M, Mingins C (1999) Design patterns and contracts. Addison-Wesley, Reading
21. Kennedy A, Syme D (2001) Design and implementation of generics for the .NET common language runtime. In: Proceedings of PLDI 2001 (ACM conference on programming language design and implementation), Snowbird, UT. Available (March 2006) at research.microsoft.com/projects/clrgen/generics.pdf
22. Kiczales G, Lamping J, Mendhekar A, Maeda C, Videira Lopes C, Loingtier J-M, Irwin J (1997) Aspect-oriented programming. ECOOP 1997 (European conference for object-oriented programming), Jyväskylä, Finland, 9–13 June 1997. In: Aksit M, Matsuoka S (eds) LNCS 1241. Springer, Berlin Heidelberg New York, pp 220–242. Available (March 2006, registration required) at http://www.link.springer.de/link/service/series/0558/papers/1241/12410220.pdf
23. Meyer B (1986) Applying 'Design by Contract'. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc; also in IEEE Comput 25(10):40–51
24. Meyer B (1988) Object-oriented software construction. Prentice Hall, Eaglewood cliffs
25. Meyer B (1991) Design by contract. In: Mandrioli D, Meyer B (eds) Advances in object-oriented software engineering, Prentice Hall, pp 1–50
26. Meyer B (1991) Eiffel: the language. Prentice Hall, Eaglewood cliffs (second printing, 1992)
27. Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice Hall, Eaglewood cliffs
28. Meyer B (2004) The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design. In: Owe O, Krogdahl S, Lyche T (eds) From object-orientation to formal methods: essays in memory of Ole-Johan Dahl. Lecture notes in computer science, vol 2635. Springer Berlin Heidelberg New York, pp 236–271. Available (March 2006) at http://se.ethz.ch/~meyer/publications/lncs/events.pdf
29. Meyer B, Arnout K (2006) Componentization: The visitor example, to appear in Computer (IEEE). Draft available (March 2006) at http://se.ethz.ch/~meyer/publications/patterns/visitor.pdf
30. Meyer B (2006) Eiffel sthe language, 3rd edn. Draft available at http://se.ethz.ch/~meyer/ongoing/etl/ (in preparation)
31. Pinto M, Amor M, Fuentes L, Troya JM (2001) Run-time coordination of components: design patterns vs. component & aspect based platforms. In: ASoC workshop (Advanced Separation of Concerns), 18–22 June 2001, Budapest. Available (March 2006) at http://www.lcc.uma.es/~lff/papers/pinto-asoc-eccop01.pdf
32. Waldén K, Nerson J-M (1995) Seamless object-oriented software architecture. Prentice Hall, Englewood cliffs